LAB 5

COMMUNICATIONS USING THE UART, PART I

INTRODUCTION

This week in lab you will fully configure the UART for serial communication between the CyBot and PC. UART stands for Universal Asynchronous Receiver/Transmitter and is a standard component of microcontrollers. A UART provides asynchronous serial data communications for compatible devices. In Lab 2, the TM4C123 microcontroller communicated with the iRobot Roomba using the Open Interface, sending commands and receiving Roomba sensor data. This communication used a UART on the microcontroller. It was configured as part of initializing the Open Interface. There are several UARTs on the microcontroller. In Labs 3 and 4, you used another UART on the TM4C123, in this case to communicate with PuTTY on the PC. You were given precompiled code in a library and used functions for initializing the UART and sending and receiving bytes of data.

Thus, so far in lab, you have not yet done any of the lower level I/O programming of the UART. I/O programming is fundamental to embedded programming. By programming the UART, you will better understand communication interfaces in general, and you will get familiar with programming any I/O module on the microcontroller. In Lab 6, you will extend Lab 5 functionality to use interrupts instead of polling as your programming method. Interrupt functionality will be useful in the lab project. In later labs, you will learn more about other I/O peripherals of the microcontroller that are connected to the sensors and servo motor and how to configure and use those I/O interfaces.

REFERENCE FILES

The following reference files will be used in this lab:

- lab5_template.c, contains a main function template as a starting point for this lab
- uart.c, contains functions that you will implement in this lab for use in future labs
- uart.h, header file for uart.c
- cyBot_uart.h, header file for pre-compiled CyBot-PC UART communication library
- libcybotUART.lib: pre-compiled library for CyBot-PC UART communication (note: must change extension of file from .txt to .lib after copying, do not open the file, download/copy only)
    - **See important NOTE below.**
- cyBot_Scan.h, header file for pre-compiled library to scan for objects
- libcybotScan.lib: pre-compiled library for scanning (note: must change extension of file from .txt to .lib after copying, do not open the file, download/copy only)
- lcd.c, program file containing various LCD functions
- lcd.h, header file for lcd.c
- timer.c, program file containing various wait commands
- timer.h, header file for timer.c
- TI Tiva TM4C123G Microcontroller Datasheet
- TI TM4C123G Register Definitions C header file: REF_tm4c123gh6pm.h
- Cybot baseboard and LCD schematics: Cybot-Baseboard-LCD-Schematic.pdf

- GPIO and UART register lists and tables: GPIO-UART-registers-tables.pdf
- Reading guides for GPIO and UART, as needed, e.g., reading-guide-UART.pdf

NOTE: Be sure to use the cybotUART files provided for Lab 5. This is a new library with new functions compared to the library provided for the UART in previous labs. Notice the functions in the header file. In Lab 5, use the new cybotUART header and library files.

The code files are available to download.

BACKGROUND

You may want to refamiliarize yourself with what you did with the UART in Labs 3 and 4.

- In Lab 3, you sent and received messages and other information between the CyBot and PuTTY on the PC.
- In Lab 4, you pressed a button and sent messages from the CyBot to PuTTY.

You can use your work from Labs 3 and 4 to get up and running in Part 1 of this lab.

Then starting in Part 2 of this lab, you will be replacing the cyBot_uart.h and libcybotUART.lib files with your own code in uart.h and uart.c files. In other words, you will be implementing the UART functions that you were using from a library in previous labs.

As you proceed through this lab, you will fully configure UART1 and write polling (busy-wait) functions for sending and receiving bytes of data. That should round-out your understanding of UART device programming.

TECHNICAL BACKGROUND

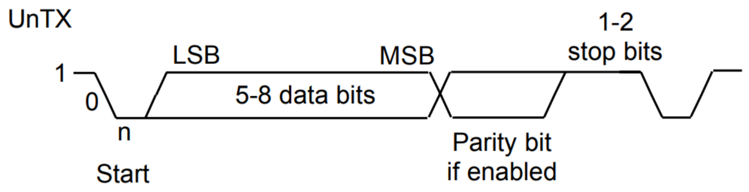
What does it mean for data transfer to happen asynchronously? Asynchronous means that the transmitter and receiver do not share a common clock. In other words, the sender transmits data using its own timing source, rather than sharing a clock with the receiver. Conversely, for synchronous data transfer, both the transmitter and receiver access the data according to the same clock. Therefore, a special line for the clock signal is required. For asynchronous communication, i.e., UART, the transmitter and the receiver instead need to agree on a data transfer speed. Both transmitter and receiver set up their own internal circuits to make sure that the data transfer follows that agreement. In addition, synchronization bits are added to the data by the transmitter and used by the receiver to correctly read the information.

Data communications are serial, which means that information is sent serially as a sequence of bits. The bits are grouped together to form serial frames. A serial frame can contain 5 – 8 bits of data, and there are synchronization bits that distinguish the start and end of a frame. There is also the option of a parity bit to provide simple error detection over a communication medium that may be noisy.

Serial communication sends one bit at a time on a single wire. The number of bits that can be transferred per second is indicated by baud rate (bit rate). This tells us the speed of communication between the transmitter and receiver.

Below is a diagram that depicts how data are transferred. UnTX is the transmit output from one of the UARTs, i.e., UARTn (where n is 0-7). LSB refers to the least significant bit of the character data, which is sent first after the start bit. MSB is most significant bit. Information about baud rate generation can be found in section **14.3.2** of the **Tiva Datasheet**. Section **14.3.3** describes in more detail how data transmission works.

Figure 14-2. UART Character Frame



Communication requires at least one transmitter and receiver, or two endpoints (one transmitting and one receiving). The CyBot is one endpoint, and your lab computer will be the other endpoint. The lab PC uses a PuTTY terminal to facilitate communications from the PC end. The PuTTY software talks directly to the serial port of your PC. As you type characters in the PuTTY terminal, ASCII characters will be sent to the serial port and over the wire in the cable.

Both endpoints must use the same frame format and baud rate. Frame format indicates the number of bits in the frame and consists of a start bit, data bits, an optional parity bit, and 1 – 2 stop bits. The standard number of ASCII character data bits in a frame can be 5, 6, 7, or 8. The parity bit can be odd, even, or none.

To generate the transmit clock in the UART (the rate at which it transmits bits on the TX wire), the UART uses a 16 MHz system clock and a clock divisor of 16 (a configurable parameter in the interface). There are two registers in a UART for configuring the baud rate: UARTn_IBRD_R and UARTn_FBRD_R. These registers combine into a 22-bit register used to calculate baud rate, i.e., a 16-bit integer part (IBRD) and a 6-bit fractional part (FBRD). Note that these represent a 22-bit binary fixed-point value. The equations for determining the integer and fractional values for these registers are given in the datasheet, textbooks, and other resources.

Code examples for sending/receiving data are in **Figure 8.73** on **page 662** in the Bai textbook and in class materials.

PRELAB

See the prelab assignment in Canvas and submit it prior to the start of lab.

STRUCTURED PAIRING

You are expected to continue to use structured pairing in this lab and in future labs. It was introduced in Lab 2.

Functional Goals for Parts of Lab:

1. Initializing the GPIO alternate function for the UART (your own code) and sending/receiving data (using precompiled library functions)
2. Initializing and configuring the UART and receiving data from PuTTY (your own code)
3. Part 2 plus transmitting characters back to PuTTY (your own code) 🡐 **Functional Milestone**
4. Sending messages to PuTTY from the CyBot, while also receiving/echoing characters from PuTTY

Part 1: UART Initialization

In Labs 3 and 4, you made use of a UART module to send information between PuTTY and the CyBot. You were given the function named `cyBot_uart_init()`. For this part, you will write code for the first half of this initialization process, which involves setting up the GPIO portion of the microcontroller. **For Part 1 of this lab, we are giving you the last half of the initialization steps in a function named** `cyBot_uart_init_last_half()`. You will replace your call to `cyBot_uart_init()` with the following code that you need to complete.

```
cyBot_uart_init_clean(); // Provided in library via cyBot_uart.h
                         // Clean UART initialization

// Complete this code to configure GPIO PORTB alternate function
// This is the first half of the UART1 initialization steps
SYSCTL_RCGCGPIO_R |= FIXME;
while ((SYSCTL_PRGPIO_R & FIXME) == 0) {};
GPIO_PORTB_DEN_R |= FIXME;
GPIO_PORTB_AFSEL_R |= FIXME; //Enable alternate function for PB pins
GPIO_PORTB_PCTL_R &= FIXME; //Force 0's at desired locations
GPIO_PORTB_PCTL_R |= FIXME; //Force 1's at desired locations
// Note: Two lines of code for PCTL can be merged into one.

cyBot_uart_init_last_half(); // Provided in library
                             // Complete UART1 device configuration
```

Your job is to replace the FIXME's with proper values.

The above code is also found in the lab5_template.c file. Make sure to copy the new libcybotUART.lib into your project.

Suggestion: You can re-use code from Lab 3 or Lab 4 to help you implement this part of the lab. The code above replaces the call to `cyBot_uart_init()`, **which was used in labs 3 and 4. Note that the cyBot_getByte() function name has changed in the new library.**

Checkpoint:

Send and receive data between the CyBot and PC, like in Lab 3 or Lab 4, using the code you developed for initializing the first/GPIO half of the UART interface. You can choose what data to send and receive.

PART 2: RECEIVE AND DISPLAY TEXT

In addition to the files that have already been provided for you, you will need to write your own **uart.c** file and **uart.h** file and the associated functions for setting up and using the UART. Separate functionalities should be in separate functions for good code quality and reusability. This means that in your uart.c file you should write separate functions for initializing/configuring the UART, sending characters, and receiving characters. Remember to use good naming conventions for function names and variables. For example, you may want to name your UART initialization function **uart_init** as there may be other initialization functions you will write in later labs that will eventually have to be used together. Minimally, we recommend the following function prototypes in your own **uart.h** file:

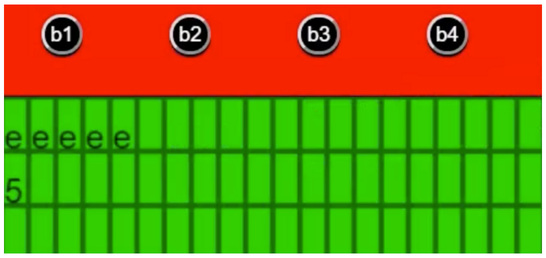
```
// UART1 device initialization for CyBot to PuTTY
// All UART1 initialization steps, both first half and last half
void uart_init(void);

// Send a byte over UART1 from CyBot to PuTTY
void uart_sendChar(char data);

// CyBot waits (i.e. blocks) to receive a byte from PuTTY
// returns byte that was received by UART1
char uart_receive(void);

// Send a string over UART1
// Sends each char in the string one at a time
void uart_sendStr(const char *data);
```

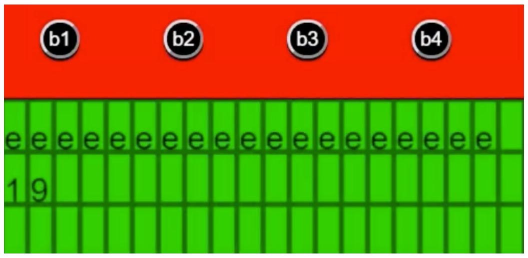
In your own **uart.c** file, implement the functions for serial communication using UART1 via Port B. The **uart.c** file includes partially implemented code for all initialization steps to guide you. You can re-use your “first half” initialization code from part 1. You will need to replace the call to the `cyBot_uart_init_last_half()` function with your own code that implements the remaining steps as outlined in `uart_init()` in **uart.c**. When you are done, `uart_init()` should be entirely your own code. You may want to test your initialization code while still using the cyBot_sendByte and cyBot_getByte functions from the library. Then when you know your initialization code is working, you can write your own `uart_sendChar()` to replace cyBot_sendByte. Test it. Then repeat for `uart_receive()`. Note: implementing the uart_sendStr() function is optional.

In order to set up UART communication, you should review section **14.4 Initialization and Configuration for UART** in the **Tiva Datasheet**. This will walk you through step by step setting up the necessary registers. Additionally, **Figure 8.73** on **page 662** of the Bai textbook provides a good example for initializing, transmitting, and receiving using the UART. **Other class resources also have examples.** You will use a baud rate of 115,200.

Make sure your **uart.c** initialization, send, and receive functions are tested and working correctly before proceeding with the remaining functionality.

Write a program that will receive characters from your computer’s PuTTY terminal window. You will buffer these characters (store them in an array) as you receive them. For debugging purposes, you should display each character on LCD line 1 as you receive it. As each character is received and placed into the buffer, a number should appear on the LCD that indicates how many characters are currently stored in the buffer, and you should print the character received. For example, this represents the LCD screen:





The program should display the entire series of characters on the LCD after 20 characters have been received, or when you press ENTER ('/r'). When you press ENTER, you should not print out the character ('\r') on the LCD. Once 20 characters have been received, clear the display prior to printing the entire series of characters on LCD line 1.

Don't worry about editing your series of characters. You don't have to manage your buffer to accommodate backspace or delete. However, if you are looking for an additional challenge, you may want to explore this feature.

CHECKPOINT:

Characters should be displayed on the LCD when 20 characters have been received (sent from PuTTY to CyBot) or ENTER is pressed.

PART 3: RECEIVE AND TRANSMIT (ECHO) CHARACTERS

In part 2, your focus was on receiving characters on the microcontroller. This part will focus on transmitting characters back to PuTTY. Extend your program from part 2 to transmit each received character back to PuTTY.

Before beginning, turn off local echo on PuTTY by selecting **Terminal → Line Discipline Options → Force off local echo and local line editing**. It may already be off. When 'enter' is pressed only a **Carriage Return ('\r')** is sent by PuTTY to the microcontroller. When transmitting back to PuTTY, you should send a **Line Feed ('\n')** following the Carriage Return. This will tell PuTTY to advance to the next line of the display.

CHECKPOINT:

The CyBot should transmit (echo) each character it receives back to PuTTY to be displayed.

PART 4: SENDING MESSAGES FROM CYBOT

After part 3, your program is receiving and echoing characters from PuTTY. Extend your program to send messages to PuTTY. Complete the uart_sendStr() function to send messages to PuTTY. You can choose what messages to send, when to send them, etc. – you may want to play around with this.

CHECKPOINT:

The CyBot should transmit each character it receives back to PuTTY to be displayed. Messages from the bot should be displayed in PuTTY.

Sketch a flowchart that illustrates the communication behavior of your program, showing the flow of control for receiving a character from PuTTY, echoing the character back to PuTTY, and sending a string (message) to PuTTY. For example, in what order do these tasks (receiving, echoing, sending) execute in your program? Be prepared to talk to your TA about the communication behavior.

DEMONSTRATIONS:

1. **Functional demo of a lab milestone –** Specific milestone to demonstrate in Lab 5 is Part 3. You should complete Part 4 if you have time, as it will be used in future labs, but it is not required for the functional demo in this lab.
2. **Debug demo using debugging tools to explain something about the internal workings of your system –** The TA will announce any specific debugging requirements at the start of lab; otherwise you will create your own debug demo based on your needs and interests in the lab.
3. **Q&A demo showing the ability to formulate and respond to questions –** This can be done in concert with the other demos.